Behavior Oriented Specification in Gist

Martin S. Feather

USC / Information Sciences Institute
4676 Admiralty Way, Marina del Rey CA 90292, USA

In *Formal program development. IFIP TC2/WG 2.1 State-of-the-Art Report*, Lecture notes in computer science, No. 755, eds.: B. Möller, H. Partsch and S. Schuman, Springer-Verlag, 1993, pp. 89-122.

Abstract

A specification language suited to specifying systems that exhibit ongoing behaviors is described and illustrated. The initial stages in the development from specifications in this language towards implementations are discussed.

1 Introduction

1.1 The Virtues of Specification

In the course of software development, a formal specification of the system being developed can be useful for two purposes:

- to serve as the contract between specifier and implementor, defining the system to be constructed, and
- to provide early feedback (i.e., prior to actual development and fielding of the complete system), so that actual needs can be accurately predicted.

Specification languages facilitate both these activities by disregarding implementation concerns, particularly those of efficiency. They are populated with constructs tailored for ease of expression, rather than ease of implementation (in contrast to programming languages, whose makeup generally reverses these priorities). This frees the specifier to more readily state *what* the system is to do, without the need to dictate *how* it is to do it. Mechanical assistance can be brought to bear in the task of deriving a satisfactorily efficient implementation from such a specification (this is the primary role of program transformation techniques). In any system development process that strives to bridge the gap between informal desires residing in peoples' heads and the programs that implement those ideas, formal specifications can serve as a crucial stepping stone.

1.2 Systems Exhibiting 'Behaviors'

In these notes I report on a specification language, **Gist**, designed for specifying systems that exhibit ongoing behaviors. Examples of such systems include a package router (a mechanism to sort postal packages into one of several bins depending upon each package's

destination), an elevator mechanism for transporting passengers to their destinations in a multi-story building, and a library database system to keep track of a lending library's books. The complexity of such systems often lies in the complexity of their ongoing interactions with their environments. As will be shown, the goal of supporting the specification of such 'behaviors' motivates the makeup of the specification language Gist. Tasks that do not exhibit such behaviors may be better specified in a different style of language. For example, sorting, searching, parsing, and unification can all be conveniently characterized as functions which compute an output given some input; for these, a purely applicative language (no side effects) seems appropriate for the purposes of specification. In contrast, Gist is founded on the notion of state, and state changes, the very antithesis of the applicative style. Nevertheless, the virtues of specification apply across many different styles of tasks: the clarity of specifications promote understandability, analysis and modification; in contrast, implementations' interwoven nature promotes efficiency at the expense of these other properties.

1.3 What Follows

These notes are structured as follows::

Section 2: a simple example on which to demonstrate the Gist style of specification.

Section 3: a more complete version of the simple example introduced in Sect. 2. This more complete version raises some issues that have relevance to formal specification.

Section 4: description of another problem, that of elevators serving passengers in a multi-story building. The Gist specification of this problem is shown, reinforcing the style of specification and use of specification language constructs introduced in the package router problem.

Section 5: a look at the initial stages in the development from a specification of the system as a whole towards the specifications of the individual components of that system, a necessary precursor to implementation of those components. This is illustrated upon the elevator problem, where the interfaces between elevator system and passengers are derived at the same time as the system-wide requirements are divided into pieces and allocated to individual components.

Section 6: a brief report upon the work that has accompanied the development and use of Gist, and the relationship of this style of specification to other approaches and techniques in the broader research community.

2 An Example - the Ideal Package Router

The first example is an idealized version of the 'package router' problem. The full form of this problem will be presented later in Section 3. The reasons for starting with an idealized version are twofold: first, it serves as a small and simple example with which to illustrate the Gist style of specification — simultaneously introducing both a complex example and the details of a specification language at the same time would probably be overwhelming; second, it reflects a plausible approach to the design of such a system — starting by describing an idealized version of the system we would like to implement.

Fig. 1. Package router

The package router is a system for distributing packages into destination bins.

Packages arrive at a source station at the top, and are fed into the network, which is a binary tree consisting of switches connected by pipes. The terminal nodes of the binary tree are bins.

Once fed from the source station into the system, each package moves downward through the pipes and switches until it reaches a bin. Each package has some particular bin as its desired destination, and the task of the system is to set the switches so that each package reaches its destination bin. The setting of a switch cannot be changed while a package is present in the switch (for fear of damaging the package).

2.1 Overall Issues of Specification

The ultimate goal is to develop the software to control the package router's switches. We begin by formally specifying the *entire* package routing system as a way to implicitly specify the behavior required of the switch controller. Our specification describes packages arriving at the source, being released into the topmost pipe, and moving through the routing mechanism via pipes and switches. Requirements such as packages getting to their correct destination bins, and switches not being changed while packages are in them, will be included. Our intent in specifying a system such as the package router is to describe all the possible and desirable behaviors of that system.

Denotation of a Gist Specification: A specification denotes a set of *behaviors*, where a behavior comprises a sequence of *transitions* between *states*. The intuition behind this terminology is explained next.

By *behavior* we mean a sequence of transitions. For example, when specifying the package router, a behavior will involve transitions such as a package arriving at the source, followed by a switch's setting changing, then the package moving into the pipe that emerges from the source, etc.

Each behavior comprises a sequence of *states*, linked by *transitions*. Think of a state as a 'snapshot' of the entire system at some moment. For example, Fig. 1 shows the system in a state in which there are several packages present (one at the source, one in a pipe, several in bins). The locations of all these packages, their desired destinations, the settings of switches, indeed, the topology of the entire package routing network, comprise the information in that state. A transition is one or more changes from one state to the next. For example, the topmost switch could change from being set to direct packages rightwards to instead direct packages leftwards. The package in the source could move downward into the topmost pipe. Within a single transition there must be at least one such change, and possibly many.

The specification denotes a *set* of behaviors, making it possible to model systems with many possible and desirable behaviors. For example, in the package router system, some behaviors would involve a package arriving at the source with the leftmost bin as its destination, while other behaviors would involve a package arriving at the source with some other bin as its destination. Another source of variation comes from the switches within the system, which are free to change their settings at any time, provided only that packages get routed to their destinations. The former example (alternative destinations of a package arriving at the source) illustrates variability in the environment of the system we are to implement, whereas the latter (switch settings) illustrates variability in the acceptable behaviors of the portion to be implemented. Both of these are manifest as multiple behaviors in the denotation of the specification.

Closed-world Nature of Specification: Notice that in describing the package router, we describe not only the portion to be implemented — switches — but also (some of) the environment in which that portion resides — packages arriving at the source and moving through the router. In this way, all constraints imposed on the implementation by its environment are made explicit, as is all information on which it can rely. We call this a 'closed system' style of specification, because the specification denotes an entire system whose behaviors can be examined without reference to a larger, unspecified, environment.

For simple input-output tasks, closed system specification is trivially achieved, for example, the specification of a square-root program stating that its input must be a non-negative number. For systems with ongoing behavior, however, there is typically much more choice in just how much of their environment to model in the specification. Often the specification can be clearer if more than the absolute minimum of the implementation's environment is included. For example, formal specification of the switch control mechanism could, as a bare minimum, comprise only the switch-setting-changing reactions required in response to the arrival of packages at various locations. It would be hard to discern from such a specification whether or not the mechanism would in fact route packages to their destination bins. In contrast, our specification will state this directly as a property required of the whole system.

Generate and Test Style of Specification: There is one more aspect that permeates the style that we recommend for systems of ongoing behavior, namely the adoption of a 'generate and test' style of specification. We find it convenient to have our specification describe a set of straightforward possible behaviors, and use constraints to rule out the undesirable ones. For example, the core of our specification of the router will describe all possible routing behaviors: those in which packages get routed to the wrong bins as well as those in which packages get routed to the right ones. We then impose constraints on these behaviors, to discard all those that do not satisfy our requirements (including the one that all packages reach their desired destination).

This generate-and-test style is not new; consider non-deterministic accepting automata. Also, consider the well-known specification of sorting as being the generation of all possible permutations of the input, followed by the discarding of all but the ordered one(s). We merely apply this same principle to the specification of ongoing behavior.

Structure of a Gist Specification: The subsections that follow introduce the details of specification in Gist. These are organized into three areas, corresponding to the three main portions of a Gist specification:

- declarations of the contents of the states that make up a specification's behaviors (Sects. 2.2 and 2.3),
- top-level statements that produce the transitions between states, and thus give rise to the behaviors denoted by the specification (Sect. 2.4), and
- constraints that serve to discard unwanted behaviors from the denotation (Sect. 2.5).

How these portions together determine the denotation of a Gist specification is presented in Sect. 2.6, and finally the uses of Gist's constructs in the package router example are summarized in Sect. 2.7.

Implementation of a Specification: A Gist specification denotes a set of behaviors. An implementation of such a specification must denote a non-empty subset of those behaviors.

2.2 Specifying What Goes in a State

What are the contents of the states that make up our specification's behaviors? For the package router, states hold information modelling the routing network itself (source, pipes, switches and bins, and how these are connected), and the packages that flow through this network (their locations and destinations).

To model the state of a system, Gist provides *objects*, and *relations* among objects. Objects in the system being specified, e.g., packages, switches and bins, are typically modelled by Gist's objects. Other information about the system, for example, where a package is currently located, the direction in which a switch is currently set, are typically modelled by Gist's relations.

It is convenient to suppose that objects are *typed*, and that types can be formed into hierarchies, as is common in many languages. For example, package and switch will be distinct types (no object can be both a package and a switch at the same time). Location is a type (the possible locations of a package), each instance of which must be exactly

one of: the source, a pipe, a switch, or a bin. Gist's types are simpler than abstract data types common to other languages — there is no encapsulation of operations with types.

Gist's relations may be of arbitrary arity — binary relations are the most common, but n-ary (for all non-negative integer values of n) are equally admissible. Relations hold among objects. Thus to represent the location of a package, we may define a binary relation `Package-Location` to hold between a package and the location of that package. Relations may be queried to find out whether or not they hold between objects. For example, if `p` denotes a package, and `l` denotes a location, then `Package-Location(p,l)` denotes a query of whether `Package-Location` holds between `p` and `l`.

We can ask *arbitrary* queries of relations — for example, given a package `p`, we may ask for any location to which is it related by the `Package-Location` relation, thus:
`any l | Package-Location(p, l) .`
Likewise, given a location `l`, we may ask for any package related to it by the `Package-Location` relation, thus: `any p | Package-Location(p, l) .`
This capacity to make arbitrary queries of relations is known as *fully associative*, and is appropriate for specification. There is no need to provide inverses of relations, since the inverse query is just as easy to issue. Of course, in *implementing* specifications, the particular relational queries that are issued, their frequency, and the desired space/time tradeoffs, will combine to determine the appropriate structures to be used to hold the information, and algorithms to retrieve that information.

More complex queries can be constructed by the usual logical connectives and quantification. For example, to ask whether *all* packages are located at the same location:
`exists l:location | for-all p:package | Package-Location(p, l) .`
(The syntax "`for-all p:package | Package-Location(p, l)`" is that of universal quantification — `p` is a quantified variable whose values may range over objects of type `package`; the predicate of the quantification — `Package-Location(p, l)` — follows these quantified variable declarations, separated from them by the "`|`" symbol. Existential quantification is written similarly, e.g., `exists l:location | ...`)

The fragments of Gist shown below declare the *types*, the *relations* among types, and some of the *constraints* on these types and relations, used to represent the objects in the package router specification. The order of declarations is not important, so I have chosen to group the type declarations first, etc.

```
type location;
type source subtype-of location;
type pipe subtype-of location;
type switch subtype-of location;
type bin subtype-of location;
type package;

relation Package-Location(package, location);
relation Package-Destination(package, bin);
relation Source-Outlet(source, pipe);
relation Pipe-Outlet(pipe, switch union bin);
relation Switch-Outlet(switch, pipe);
relation Switch-Setting(switch, pipe);
```

```
constraint Unique-Source
  not exists s1:source, s2:source | s1 /= s2;

constraint Unique-Source-Outlet
  not exists p1:pipe, p2:pipe |
        p1 /= p2 and
        Source-Outlet(the source, p1) and
        Source-Outlet(the source, p2);

constraint Switch-Set-To-Outlet
  for-all s1:switch, p1:pipe |
    Switch-Setting(s1, p1) implies
    Switch-Outlet(s1, p1);

constraint Unique-Switch-Setting
  not exists s:switch, p1:pipe, p2:pipe |
        p1 /= p2 and
        Switch-Setting(s, p1) and
        Switch-Setting(s, p2);
```

In the above, `location`, `source` etc., are defined as types. The states within the specification's behaviors may contain objects of these types. The types `source`, `pipe`, `switch` and `bin` are further declared to be subtypes of type `location`, so that any object that is one of those types is also of type `location`.

`Source-Outlet`, `Pipe-Outlet` etc., are defined as relations. The states within the behaviors denoted by the specification may contain instances of these relations holding among objects. In declaring a relation, the types of the objects it may relate are given. For example, relation `Source-Outlet` is declared to relate objects of type `source` to objects of type `pipe`; only objects of these types may be related by `Source-Outlet`.

`Unique-Source`, `Unique-Source-Outlet`, `Switch-Set-To-Outlet` and `Unique-Switch-Setting` are defined as constraints. Constraints impose restrictions on the behaviors that the specification can denote. The `Unique-Source` constraint prohibits there from being two distinct objects of type source. The `Unique-Source-Outlet` prohibits there from being two distinct pipes to which the the source is related by relation `Source-Outlet`. The `Switch-Set-To-Outlet` constraint prohibits each switch's setting from being to anything other than one of its outlets (in other words, a switch can only be set to direct packages towards one of its outlets, not to some arbitrary location anywhere in the router). Constraints such as these may seem trivially obvious to anyone looking at the picture in Fig. 1, but it is necessary to state them explicitly in the formal specification. For brevity, I have omitted similar constraints on the other relations (e.g., uniqueness of location and destination for each package, the acyclic topology of the tree, etc.).

In general, a Gist constraint is expressed as a predicate; in order for a behavior to be allowed, every state of that behavior must satisfy every constraint. A behavior in which one (or more) of the states does not satisfy a constraint is said to be 'anomalous', and

is not included in the denotation of the specification; this will be discussed further in Sect. 2.6. Note that while these constraints restrict the topology of the router network within each state, they do not prohibit the topology from changing from state to state — for example, I have made no mention of any constraint prohibiting the source from being connected to one pipe in one state, and a *different* pipe in the next state! One of Gist's underlying assumptions is the so-called 'frame' property — in a transition from one state to the next, changes take place to only the information explicitly changed; all else remains the same. Thus if nothing in the specification ever explicitly changes the connectivity of, for example, the source to a pipe, we are assured that it will remain constant throughout every behavior. We can, if need be, write constraints that span multiple states, and so prohibit certain transitions that we wish to exclude. We will see an example of one of these constraints later.

Remarks: It is obvious from the above fragment of the Gist specification that there are many opportunities for syntactic abbreviations. In these notes, I generally avoid the use of abbreviations, favoring a somewhat verbose but simple form of the language.

It is also obvious that the specification of a common structure such as a binary tree is likely something that we would wish to reuse, and so it would be worthwhile to provide generic such definitions, and allow their instantiation to the particular task in hand. Again, for simplicity this practice has not been followed in these notes — instead, the basic form of Gist is used, with an emphasis on presenting the fundamentals of behavior oriented specification.

2.3 Implicit Information — More of What Goes in States

The relations introduced earlier (`Source-Outlet`, etc.) must have their values explicitly inserted and removed by the specification (we will see shortly how this is done). Gist provides another class of relations, called *implicit* relations, whose values are *implicitly* defined in terms of other information in the specification. An implicit relation is defined by means of a predicate expressed over the objects that could participate in the relation — in a given state, the relation holds among those objects if and only if the predicate is true of those objects when evaluated in that state. The fragment of Gist below shows the definitions of three such relations:

```
implicit relation Immediately-Below(lower:location, upper:location)
  iff Source-Outlet(upper, lower) or
      Pipe-Outlet(upper, lower) or
      Switch-Outlet(upper, lower) ;

implicit relation Somewhere-Below(lower:location, upper:location)
  iff Immediately-Below(lower, upper) or
      exists intermediate:location |
        Somewhere-Below(lower, intermediate) and
        Immediately-Below(intermediate, upper) ;
```

```
implicit relation Dynamically-Immediately-Below
                                  (lower:location, upper:location)
  iff Source-Outlet(upper, lower) or
      Pipe-Outlet(upper, lower) or
      Switch-Setting(upper, lower)
```

The binary relation `Immediately-Below` is defined to hold between two objects of type location, `lower` and `upper`, if and only if either the relation `Source-Outlet` holds between `upper` and `lower`, or relation `Pipe-Outlet` holds ... etc. In other words, `Immediately-Below` captures the connectivity of the package router structure of source, pipes, switches and bins.

Similarly, `Somewhere-Below` is defined to hold between objects `lower` and `upper` if and only if either `Immediately-Below` holds between `lower` and `upper`, or there exists some location `intermediate` for which `Somewhere-Below` holds between `lower` and `intermediate`, and `Immediately-Below` holds between `intermediate` and `upper` — that is, `Somewhere-Below` is the transitive closure of `Immediately-Below`. For recursively defined relations such as `Somewhere-Below`, Gist assumes least fixpoint semantics where the ordering is inclusion of relations.

Finally, relation `Dynamically-Immediately-Below` is similar to `Immediately-Below`, except for the case of switches, where it uses `Switch-Setting` rather than `Switch-Outlet` to determine that only the pipe to which that switch is currently set is in the `Dynamically-Immediately-Below` relation with that switch.

Implicitly defined relations can be queried just as regular, non-implicit, relations, for example, within the definition of `Somewhere-Below` is a query of `Immediately-Below`.

Remarks: It is purely an implementation concern as to when and how to compute the values of implicit relations (at one extreme we might compute their values only when necessary, i.e., on evaluating a specific query of such a relation; at the other extreme we might store and continually update their values so as to keep them up-to-date as changes occur to the information in terms of which they are defined). The various tradeoffs between such implementation alternatives may be of crucial importance in achieving a sufficiently efficient implementation, but have no relevance to our aim of *specification*.

2.4 Specifying Transitions Between States

So far we have considered only the information that resides in individual states. What happens in the transitions between states?

Primitive Changes in Transitions: Since the contents of a state comprise typed objects and relations among those objects, transitions are comprised of a set of primitive changes to these. In Gist, the allowed primitive changes are to:

insert that a relation holds among objects (if the relation already holds among those objects, then such an insertion is legal, and is a no-op, i.e., makes no change),

remove that a relation holds among objects (if the relation does not hold among those objects, then such a removal is legal, and is a no-op, i.e., makes no change),

create a (new) object of a given type (this necessarily results in an object that is guaranteed to have not existed in any prior state), and

destroy an existing object (destruction of an object automatically removes all the relations holding of that object, thus saving the specifier from the tedium of explicitly writing all the removals).

For example, `insert Package-Location(p, l1)` is a primitive change, inserting the relation `Package-Location` between the package referred to by `p` and the location referred to by `l1`.

Simultaneous Primitive Changes in Transitions: In a transition from one state to the next, at least one such primitive change, and possibly many, take place. In the case of multiple changes taking place in the same transition, they are assumed to take place simultaneously, i.e., there is no 'intermediate' state in which some but not all of those transitions have taken place. Gist's `atomic` construct is used to group primitive changes to have them occur simultaneously, for example,
`atomic {insert Package-Location(p, l1); remove Package-Location(p, l2)}`
simultaneously inserts that the relation `Package-Location` holds between package `p` and location `l1`, and removes that it holds between `p` and `l2`.

Gist requires that following a transition, all the 'inserted' relations hold in the resulting state (e.g., following a transition in which `insert Package-Location(p, l)` occurs, `Package-Location(p, l)` holds), all the 'removed' relations do not hold in the resulting state, all the 'created' objects exist, and all the 'destroyed' objects do not exist. Thus, Gist does not allow the simultaneous insertion and removal of the same relation among the same objects, e.g.,
`insert Package-Location(p, l)` and
`remove Package-Location(p, l)` cannot both occur as primitive changes within a single transition. We say that such a transition is 'anomalous', and is not allowed within any behavior denoted by the specification. We will return to the precise meaning of anomalous transitions shortly (Sect. 2.6). Likewise, it is anomalous for a transition to insert some relation involving an object that does not exist in the resulting state (which could occur either because the object is destroyed in that transition, or because it didn't exist beforehand, and is not created by that transition), e.g.,
`insert Package-Location(p, l)` and
`destroy p` cannot both occur as primitive changes within a single transition. Finally, it is anomalous to create and destroy the same object in the same transition.

Transitions in the Package Router: For the package router, transitions include arrival of a package at the source, movement of a package through the router, and changing of switch settings.

Arrival of a package at the source can be modelled by the *creation* of a (new) object of type package, whose location is set to the source, and whose destination is set to some bin. Since a package's location and destination are represented by the relations `Package-Location` and `Package-Destination` respectively, setting these values is done by inserting that the relation `Package-Location` holds between the package and the

source, and inserting that the relation `Package-Destination` holds between the package and some bin, as follows:

```
atomic { create p:package;
         insert Package-Location(p, any source);
         insert Package-Destination(p, any bin)   }
```

In the above, `create p:package` creates a (new) object of type package locally named `p`; the first of the `insert` statements inserts the relation `Package-Location` between the new package and any object of type source — since there's only one source in the package router, this is unambiguous; the second of the `insert` statements inserts the relation `Package-Destination` between the new package and any bin. Since there are many objects of type bin, there are many possible evaluations of this statement — one for each bin! The specification denotes a different behavior for each possible evaluation. We refer to this multitude of possibilities as *nondeterminism*. The expression `any bin` has multiple possible values (any one of the several bins). When such an expression is used within a primitive statement, as in `insert Package-Destination(p, any bin)` above, the result is a different behavior for each choice. An expression may have *no* values, for example, if in some state there are no packages located at location `l`, say, then
`any p:package | Package-Location(p, l)`
is an expression without *any* values. A primitive statement that used the value of such an expression, would be 'anomalous', e.g.,
`insert Package-Location( any p:package | Package-Destination(p, l1), l2)`
would be anomalous if there were no package with destination `l1`. Again, we defer further discussion of 'anomalous' transitions until Sect. 2.6.

Top-level Statements Causing Transitions: A Gist specification comprises declarations of types and relations, constraints that prohibit certain states from arising, and, as we focus on now, top-level *statements* that cause transitions to take place, i.e., generate the specification's behaviors.

A Gist specification without any top-level statements would have as denotation a set of behaviors each comprising a single state, e.g., the state of the package router at a single moment in time. It is by including top-level statements that a Gist specification denotes behaviors with multiple states. Each top-level statement in the specification begins execution in the starting state. As they execute, they contribute the primitive changes that comprise the transitions in the behavior. When all these statements have completed their execution, the behavior is complete. For systems such as the package router, we often make use of non-terminating statements to model the potentially infinite behavior of the router, which is supposed to continue to route packages for ever.

In the case of multiple statements executing concurrently, Gist assumes an interleaving and merging semantics, that is, in the transition from the current state to the next, at least one, and possibly many, of the currently executing statements is/are executed until it/they contribute a primitive change (or, through execution of an atomic statement, a set of primitive changes); all the contributed primitive changes are unioned together to form the transition to the next state. Unions of primitive changes that constitute 'anomalous' transitions (e.g., the simultaneous insertion and removal of the same relation among the same objects) are omitted from the denotation of the specification. In the case of

multiple statements, every possible non-anomalous combination of at least one of those statements is taken, producing a set of possible next transitions; this is manifest as a branching in the behavior tree — from the current state, there may be many possible continuations. Note that several statements can contribute to the same transition, i.e., Gist not only interleaves multiple statements, but also merges them.

Top-level Statements in the Package Router: The sources of activity in the package router network are packages arriving at the source and moving through the router, and switches in the router changing their settings.

Arrival of packages at the source is modelled as a continuous loop in which there is choice of whether or not to create a new package at the source:

```
loop while true
do choose { null;
            atomic { create p:package;
                     insert Package-Location(p, any source);
                     insert Package-Destination(p, any bin) } }
```

Gist's `loop` statement is similar to that of conventional languages — it directs the execution of the loop body. Other conventional constructs that Gist uses include the conditional statement:
`if <predicate> then <statement> else <statement>` ,
the sequential statement:
`begin <statement> ; <statement> ; ... end` ,
the case statement, and so on. Ultimately these bottom-out at Gist's primitive statements that denote transitions, which make changes to the relations among objects, and create or destroy objects.

There are couple more Gist-specific statements used in the above:
The statement `null` means to do nothing.
The statement `choose { ... }` is another compound statement, offering the nondeterministic choice of any of its constituent statements. In this case, the choice is between doing nothing, and creating a new package at the source.

By placing the above loop statement as a top level statement within the specification, its execution commences in the starting state of the whole system.

Movement of packages through the router network is modelled as follows:

```
demon Move-Package-Through-Network(p:package)
  when exists loc:location, next-loc:location |
         Package-Location(p, loc) and
         Dynamically-Immediately-Below(next-loc, loc)
  do choose { null;
              atomic{ remove Package-Location(p, loc);
                      insert Package-Location(p, next-loc) } }
```

Instances of Gist's `demon` construct, such as this, are placed at the top level of the specification. They contribute to the transitions between states in the following manner:

in every state, for every instantiation of appropriately typed objects in that state to parameters of the demon such that the demon's predicate (which follows the '`when`' keyword) is true, the demon's statement (which follows the '`do`' keyword) begins execution, contributing its primitive changes to the transition that emerges from that state.

In the case of the `Move-Package-Through-Network` demon, there is one parameter `p`, of type `package`. Thus, in every state, for every instantiation of `p` to some object of type `package` existing in that state, the predicate is evaluated, and if true, the statement begins execution. In this case, the predicate is that package `p` be located at some location `loc` and there's another location `next-loc` dynamically immediately below `loc`. The activity is a choice of nothing, or the transition that models the movement of `p` from `loc` to `next-loc`. Following the style outlined earlier, this transition is expressed as an atomic containing a set of changes to be made simultaneously, the changes being to remove that the relation `Package-Location` holds between the package and its current location `loc`, and insert that it holds between the package and the next location `next-loc`. Recall that bins never have any location below them, so once a package reaches a bin, it is not moved by this demon.

The changing of switch settings is modelled similarly, as follows:

```
demon Change-Switch-Setting(s:switch)
  when exists current-setting:pipe, other-setting:pipe |
         Switch-Setting(s, current-setting) and
         Switch-Outlet(s, other-setting) and
         other-setting /= current-setting
  do choose { null;
              atomic{ remove Switch-Setting(s, current-setting);
                      insert Switch-Setting(s, other-setting)     } }
```

By placing the declarations of these demons in the specification, along with the looping statement modelling package arrival at the source, we now have modelled all the activity of the package router.

Remark: We could equivalently have used a top-level statement in the form of an infinite loop over all the switches, rather than a demon, to do this changing of switch settings. The choice of construct is a matter of convenience and style.

2.5 Constraints on Behaviors

We have already introduced some constraints on the possible behaviors of the package router — for example, that the switches be set to one of their outlet pipes. There are other constraints that must be added, namely that packages eventually reach their destination bins, and that no switch setting be changed while there is a package located in that switch. These may be expressed as follows:

```
constraint Packages-Reach-Their-Destinations
  for-all p:package, destination:bin |
    Package-Destination(p, destination) implies
```

```
    ( exists s:past-present-or-future-state |
                  Package-Location(p, destination) as-of s );

constraint Switch-Setting-Not-Changed-While-Package-Present
  for-all s:switch, p1:pipe, p2:pipe | p1 /= p2 and
    Switch-Setting(s, p1) and ( Switch-Setting(s, p2) as-of next )
    implies Empty-Switch(s);

implicit relation Empty-Switch(s:switch)
  iff not exists p:package | Package-Location(p, s) ;
```

The expression of each of these constraints makes use of the ability to make queries about states anywhere in the current behavior, not just the state in which the query itself is being evaluated. In particular, `exists s:past-present-or-future-state | ...` quantifies over all states in the behavior, allowing us to ask whether there exists some state `s` in which the package is located at its destination in that state ('`as-of s`' causes the query that precedes it to be evaluated in the state s). In the second constraint, `next` is used to refer to the immediately following state. The constraint requires that in any transition in which a switch's setting changes (recognized by it being set to pipe `p1` in one state, and a different pipe `p2` in the next state), then the switch must be empty in the first of those states. Note that while this disallows changing setting if a package is present at the start of that transition, it does allow changing if a package enters the switch simultaneously with the setting change — if the latter is also to be prohibited, simply extend the right hand side of the implication to read:
`Empty-Switch(s) and (Empty-Switch(s) as-of next)`.
The definition of `Empty-Switch` is given as an implicit relation.

Many specification languages provide similar abilities, in the form of *temporal operators* such as eventually and next (state). Gist is not particularly progressive in this regard, merely making use of this capability as useful for specification.

Remark: Since there is only one place in the specification where switches change their settings, we could equally well have written this as a precondition on the switch setting changing, thus:

```
demon Change-Switch-Setting(s:switch)
  when exists current-setting:pipe, new-setting:pipe |
         Switch-Setting(s, current-setting) and
         new-setting /= current-setting
  do choose { null;
              precondition not exists p:package | Package-Location(p, s)
                atomic{ remove Switch-Setting(s, current-setting);
                        insert Switch-Setting(s, new-setting) } }
```

2.6 Denotation of the Package Router Specification

We have completed all the essential elements of the idealized package router specification:

- Type and relation declarations define the possible objects and relations among objects that can occupy states;
- Constraints prohibit undesired states from occurring; and
- Top-level statements, which begin execution in the initial state, and demons, which begin execution of their statements whenever their predicates are true, provide the transitions from one state to the next.

Thus in our example, the loop statement that creates packages at the source, together with the demons that cause switches to change their settings and packages to move through the network, generate the possible behaviors of the package router specification. As stated earlier, Gist semantics interleave and merge these statements' primitive changes to form the transitions between states (e.g., a switch changes its setting, followed by creation of a package, followed by that package moving into the topmost pipe simultaneously with a switch changing its setting, and so on). As in this system, there are typically many ways of performing such interleavings and mergings, offering a very potent source of nondeterminism. The various sources of nondeterminism — expressions with multiple values used within primitive statements, the `choose` statement, and finally the interleaving and merging of the multiple statements and demons, all combine to generate a multitude of possibilities, in line with the 'generate' aspect of our generate-and-test philosophy.

By definition, the denotation of a Gist specification is the subset of the possible behaviors which are not 'anomalous'. Recall that a behavior will be anomalous if:

- any of its states violates a constraint,
- any of its transitions' activities violate any of their preconditions or postconditions (semantically, these can be treated as 'point constraints'),
- any of its transitions involve the simultaneous insertion and removal of the same relation among the same objects, or
- any of its transitions involve the simultaneous creation and destruction of the same object, or
- any of its transitions involve the insertion of a relation on an object that does not exist in the end-state of that transition.

This is illustrated diagrammatically in Fig. 2, where the behaviors are drawn as a tree whose root (at the top) is the common initial starting state, and whose paths from root to leaf correspond to behaviors; the result of 'pruning' to meet the constraints is shown on the right, where all the behaviors containing any bad states (states that violate one or more constraints) have been omitted. Observe that sometimes whole subtrees get lopped off by this process.

Thus our idealized package router specification denotes all possible behaviors in which switches change or retain their settings (changing only when no packages are present), while packages get created at the source and move downwards through the router to eventually arrive at their destination bins (such that when moving out of a switch, a package moves to the pipe to which that switch is currently set).

This illustrates the 'generate-and-test' style of specification — nondeterminism is used to straightforwardly express a range of possible behaviors, while constraints are used to discard those behaviors which are physically impossible (e.g., a switch being set to a pipe

Fig. 2. Constraints and pruning

that is not one of its outlets), or undesirable (e.g., packages not eventually reaching their destinations).

2.7 Summary of Constructs Used in the Specification

The various Gist features used in the specification of the idealized package router were:

types — used to represent the different kinds of objects in the package router (e.g., locations, packages).

relations — used to represent information about the state of the objects in the system (e.g., package locations, switch settings).

implicitly defined relations — used to define information in terms of other information (e.g., to define whether one location is 'below' another).

primitive statements — used to express the changes that take place in moving from one state of the system to the next (e.g., movement of a package; change of a switch setting).

compound statements — used to group together statements. We saw a simple loop statement, an atomic statement, and a choose statement. Sequentiality is provided by `begin ... end` blocks of statements (not used in the examples of this paper). The atomic statement groups together statements so that execution of the atomic

statement simultaneously executes all of its primitive statements within the same transition (e.g., creation of a package object simultaneously with the assertion of that package's location and destination). Thus the atomic statement cannot include sequential statements, nor most loop statements (however, simple forms of loops that perform an atomic action on each member of a set of objects can be so included, meaning the action is performed simultaneously on each of the objects in the set).

demons — used to cause some activity to take place whenever some condition is true (e.g., a package to move to the next location).

temporal reference — used to extract information from arbitrary states within the current behavior (e.g., to ask whether a package is ever located at its destination).

nondeterminism — used to generate a number of possible behaviors (e.g., a package is created with any one of the bins as its destination; a switch's setting may be left alone, or changed; a package not already at a bin may stay in place or move to the next location down in the router).

constraints — used to restrict the behaviors denoted by the specification (e.g., to just those in which packages eventually reach their destinations). In conjunction with nondeterminism, they support a 'generate and test' style of specification.

3 The Complete Package Routing Problem

The package router problem was constructed by representatives of the process control industry to be typical of their real-world applications. A study of various programming methodologies was done using this as the comparative example [22]. The complete description of the problem is as follows:

> The package router is a system for distributing packages into destination bins.
>
> A source station at the top feeds packages one at a time into the network, which is a binary tree consisting of switches connected by pipes. The terminal nodes of the binary tree are the destination bins.
>
> When a package arrives at the source station, its intended destination (one of the bins) is determined. The package is then released into the pipe leading from the source station. For a package to reach its designated destination bin, the switches in the network must be set to direct the package through the network and into the correct bin.
>
> Packages move through the network by gravity (working against friction), and so steady movement of packages cannot be guaranteed; so they may "bunch up" within the network and thus make it impossible to set a switch properly between the passage of two such bunched packages (a switch cannot be set when there is a package or packages in the switch for fear of damaging such packages). If a new package's destination differs from that of the immediately preceding package, its release from the source station is delayed a (pre-calculated) fixed length of time (to reduce the chance of bunching). In spite of such precautions, packages may still bunch up and become mis-routed, ending up in the wrong bin; the package router is to signal such an event.
>
> Only a limited amount of information is available to the package router to effect its desired behavior. At the time of arrival at the source station but not

thereafter, the destination of a package may be determined. The only means of determining the locations of packages within the network are sensors placed on the entries and exits of switches, and the entries of bins; these detect the passage of packages but are unable to determine their identity. (The sensors will be able to recognize the passage of individual packages, regardless of bunching).

For the purposes of introducing Gist's features, we omitted some of the details of the complete version of the package router problem. We now discuss these details, and the ways in which they complicate the specification.

3.1 Accommodating Misrouting of Packages

The complete problem makes it clear that there are circumstances in which the correct routing of every package cannot be guaranteed. Hence the constraint `Packages-Reach-Their-Destinations` that we included in our idealized version should not be included in the complete version, because it would make the implementation task *impossible!* In essence, there is nothing the implementation can do, given that it controls only the setting of switches, to correctly route every package in the face of the potential irregularity of package movement.

Instead, we would have to adopt a weaker constraint, for example:

> Prohibit an empty switch being set (or left set) the wrong way with respect to a package in the pipe entering that switch.

This can be expressed in Gist as follows:

```
implicit relation Switch-Set-Wrong-Way-For-Package(s:switch, p:package)
  iff exists b:bin, pi:pipe |
        Package-Destination(p, b) and Somewhere-Below(b, s) and
        Switch-Setting(s, pi) and not Somewhere-Below(b, pi);

constraint Prohibit-Malicious-Empty-Switch
  not exists s:switch, p:package, b:bin, pi:pipe |
     Package-Location(p, pi) and
     Pipe-Outlet(pi, s) and
     Empty-Switch(s) and
     ( Switch-Set-Wrong-Way-For-Package(s, p) as-of next )
```

The first definition is of a relation that holds of a switch and package if the switch lies on route to the package's destination bin, but is currently set the wrong way. The second defines a constraint that prohibits the transition from a state in which there's a package immediately above an empty switch, to a state in which the package is in the switch, but the switch is set the wrong way for that package.

In the problematic cases, when packages get bunched up, this constraint favors the correct routing of the *first* package of the bunch. This might be inappropriate, e.g., favoring the majority of packages sharing the same destination bin (when a bunch of more than two packages is formed) might be preferred. Presumably the 'correct' strategy for routing will depend on the distribution of package arrivals (their times and destinations) and package movement through the network.

Remark: The above constraint has been carefully crafted to identify the very last opportunity to change (or leave correctly set) a switch's setting for an approaching package, based upon the constraint that prohibits changing the switch setting once there's a package actually present in the switch. This careful crafting is indicative that either the constraint itself is too 'implementation' oriented, or that there should be a better way of specifying it (which perhaps Gist is not capable of stating). This remains (in my mind) an interesting open issue, as does the connection to real time specification.

In any event, the complete specification calls for the signalling of misrouting, which can be easily specified thorough a Gist demon, thus:

```
demon Signal-Misrouting-On-Arrival(p:package)
  when exists b:bin | Package-Location(p, b) and
                      not Package-Destination(p, b)
  do Signal-Misrouting(p, b)
```

where we assume that `Signal-Misrouting(p, b)` models the activity of signalling.

3.2 Availability of Information

Note that the `Signal-Misrouting-On-Arrival` demon above queries a package's destination (`Package-Destination(p, b)`) at the time the package reaches a bin. This is perfectly acceptable for specification purposes — the demon merely specifies *when* signalling takes place. However, for implementation purposes, it may not be possible to query the destination of a package arriving at the bin, and an implementation would have to find some other way of computing the information necessary to know when to do the signalling. Indeed, in the description given earlier of the complete package routing problem, this is precisely the case. An implementation would presumably read packages' destinations as they arrive at the source, and keep track of them as they move down through the router, so as to be able to know when to signal misrouted arrivals.

The general point is that in a formal specification, we separately state the behaviors required of the router, and the limits on availability of information. In specifying the behaviors required, we are *not* constrained in any way by these limits; we can continue to express our behavioral specification in terms of information drawn from anywhere in the system.

Expressing the limits on availability of information is relatively easy. The implementation is allowed to know the structure of the package router (i.e., type `location` and its subtypes `source`, `pipe`, `switch` and `bin`, relations `Source-Outlet`, `Pipe-Outlet`, `Switch-Outlet` and `Switch-Setting`). Additionally, it can know of the arrival of a package at the source, and, in the state following that transition, the destination of such a package, i.e., it can know the value of the bin `b` for which `Package-Destination(p, b)` is true, for just-arrived package `p`. Finally, it can know of the passage of packages past sensors, i.e., the occurrence of transitions when a package moves into / out of switches, and into bins (and, presumably, the identity of those locations).

In implementing the control mechanism we would be concerned with how to deduce from this available information the information necessary to set switches as required and issue misrouted signals. As we have stated before, the purpose of the specification is

merely to state the requirements, not to do this implementation (however, see the next section for related comments).

3.3 The Intertwining of Specification and Implementation

Balzer and Swartout have pointed out [34] that the natural-language statement of the package router problem already mixes implementation with specification. An example is the recognition that misrouting is inevitable given the vagaries of package movement through the router. Similarly, the sensors provide enough and just enough information to permit an implementation of the switch controller. Their conclusion is that the processes of 'specification' and 'implementation' are not as separable as we would like to believe, and that development must take into account, in fact, support, such interleaving. This does not detract from the need for specification languages — we still need to be able to represent specifications and their intermediate versions as implementation concerns are taken into account.

4 The Elevator Example

I now consider another problem, that of elevators ('lifts' in British terminology) in a multi-story building, used by passengers to get to their destination floors. In a very abstract sense, this is similar to the package router problem, insofar as they both concern the transportation of objects to their destinations. Thus by presenting (portions of) the Gist specification of elevators, this should reinforce the message of how to use Gist's constructs for specification. Additionally, this example will serve as illustration for the following section's consideration of the development from specification of the system as a whole towards specification of the individual pieces, and the interfaces among them.

Following the manner in which we specified the package router, we may specify the elevator controller by specifying a closed system involving the activities of elevators (moving between floors and opening/closing their doors) and passengers using those elevators (entering and exiting). The interleaving of these capabilities denotes a large set of transportation behaviors. Constraints prune this set, eliminating both physically impossible behaviors (e.g., ones involving passenger entry through a closed door), and undesirable behaviors (e.g., a passenger getting farther from his/her destination).

```
type floor = 1..topfloor;
type elevator;
type passenger;

relation AtFloor(elevator, floor);
relation DoorsOpen(elevator);
relation Location(passenger, floor union elevator);
relation Destination (passenger, floor);

loop while true
  do choose { null ;
```

```
              atomic { create p:passenger;
                       insert Location(p,any floor);
                       insert Destination(p,any floor) } };

demon passenger-activity(p:passenger)
  when true
  do choose
       { null;
         precondition exists f:floor, e:elevator |
                       Location(p, f) and
                       (not Destination(p, f)) and
                       AtFloor(e, f) and DoorsOpen(e)
           atomic { insert Location(p, e); remove Location(p, f) };
         precondition exists f:floor, e:elevator |
                       Location(p, e) and DoorsOpen(e) and
                       AtFloor(e, f)
           atomic { insert Location(p, f); remove Location(p, e) } };

demon elevator-activity(e:elevator)
  when true
  do choose
       { null ;
         precondition not DoorsOpen(e) insert DoorsOpen(e) ;
         precondition DoorsOpen(e) remove DoorsOpen(e) ;
         precondition (not DoorsOpen(e)) and
                      exists f:floor | AtFloor(e, f)
             choose
               { atomic {insert AtFloor(e, f+1); remove AtFloor(e, f)};
                 atomic {insert AtFloor(e, f-1); remove AtFloor(e, f)}}
             };

implicit relation FLocation(p:passenger,f:floor)
  iff Location(p, f) or
      exists e:elevator | Location(p, e) and AtFloor(e, f);

constraint no-farther-from-destination
  not exists p:passenger |
    Abs( FLocation(p,?) - Destination(p,?) ) <
    Abs( FLocation(p,?) - Destination(p,?) ) as-of next;

constraint capacity-of-elevator
  not exists e:elevator | Size({p:passenger | Location(p,e)}) > 10;

constraint each-passenger-has-unique-destination-and-location
  all p:passenger | Size({f:floor | Destination(p, f)}) = 1 and
                    Size({l:floor union elevator | Location(p, l)}) = 1;
```

```
constraint each-elevator-always-at-unique-floor
  all e:elevator | Size({f:floor | AtFloor(e, f)}) = 1
```

The above specifies the types that are used to model objects of the elevator world, namely floors (represented as integers in the range *1* to *topfloor*), passengers and elevators, and relations used to model the relationships among those objects, e.g., the floor at which an elevator is located `AtFloor`, the location of a passenger, either a floor or an elevator `Location`.

Passenger appearance is modelled by an infinite loop that may choose to create a new passenger at a floor.

Passenger activity is modelled by a demon that, for each passenger, chooses either to do nothing, to cause the passenger to enter a elevator (provided that the passenger is at a floor where the elevator is also located, and that the elevator's doors are open), or to cause the passenger to exit an elevator (provided that the passenger is inside the elevator, and the elevator's doors are open).

Passengers objects never disappear from the system — they simply remain at their destination floors, once they finally get there. If this is deemed stylistically inappropriate, it would be a simple matter to add a demon that destroyed such objects.

Elevator activity is modelled by a demon that, for each elevator, chooses either to do nothing, to open the doors (provided the doors are closed), to close the doors (provided the doors are open), or to move the elevator up or down one floor (provided the doors are closed). Note that because the relation `AtFloor` is declared to hold between an object of type `elevator` and an object of type `floor`, and because type `floor` is declared to be an integer in the range `1..topfloor`, elevator activity is implicitly constrained to prohibit movement of an elevator to a floor not within that range.

The denotation of the specification is all possible interleavings of these activities, pruned by all the constraints. Note the use of a mixture of preconditions and constraints to specify restrictions on the possible relationships, e.g., that each passenger is always located at exactly one location, either a floor or an elevator. The possible transportation behaviors, even if quite safe, include many undesirable behaviors from the point of view of getting passengers rapidly to their destinations. Hence some constraints are present to further restrict the system behaviors to only those in which acceptably efficient transportation of passengers occurs, e.g., ones in which passengers never get *farther* from their destinations. This last works by prohibiting the existence of a passenger whose distance from destination in one state is less than that in the next state (note the use of `FLocation` to cause this calculation for a passenger inside an elevator to use the elevator's floor location to do the calculation).

The above specification makes occasional use of a few constructs not introduced in the package router example, which need explanation:
functions, `Size`, `Abs` `+` and `-`,
set-former notation, e.g., { `f:floor | Destination(p, f)` }, denoting the set of all floors `f` which are destinations of passenger `p`, and
retrieval of values from relations, e.g., `Destination(p,?)` is an expression whose value is the floor related to passenger `p` by the relation `Destination`.

4.1 Summary of Constructs Used in the Elevator Specification

Gist's features were used in the elevator specification in a similar manner to the way in which they were used in the package router. Briefly:

types — used to represent the different kinds of objects (e.g., floors, passengers).
relations — used to represent information about the state of the objects in the system (e.g., passenger locations, status of elevator doors).
implicitly defined relations — used to define the relation `FLocation` between a passenger and a floor to be the floor at which either the passenger is directly located, or where there's the elevator the passenger is inside.
primitive statements — used to express the changes that take place in moving from one state of the system to the next (e.g., change of location of a passenger; change of status of an elevator's doors).
compound statements — used to group together statements. Again, loop, conditional, choice and atomic statements came into play in this specification.
temporal reference — used in expression of the constraint that passengers not get farther from their destination floors.
nondeterminism — used to generate a broad range of transportation behaviors (e.g., choice of movement of elevators in either direction, choice of passenger activity).
constraints — used to restrict the behaviors denoted by the specification to both physically possible ones (e.g., in which passengers don't pass through closed elevator doors) and desirable ones (e.g. that passengers never get farther from their destinations).

5 Initial Stages of Developing an Implementation

The elevator specification denotes behaviors required of the closed system, comprised of elevators and passengers. A typical implementation of this system will combine several *components*, an elevator controller, in charge of all the elevators, and individual passengers. We can use the system specification as an *implicit* specification of those components, namely those which, in combination, achieve the specified closed system behaviors. Thus if the ultimate task is to develop the implementation of one or more of the components, then an important step in that development will be the decomposition of the closed system specification into explicit specifications of individual components. Note that the closed system specification may be neutral with respect to which of its components we are to implement — for example, our task might be to develop an implementation of the elevator controller, or equally well to develop the implementation of individual passengers! The latter possibility would make sense if developing a users' guide for passengers, or if developing an implementation for robots that are to use elevators in order to travel between floors in a multi-story building.

When system-wide constraints occur in a Gist specification, the decomposition step must split those constraints into pieces such that each piece can be assigned as the 'responsibility' of individual components. Thereafter, an implementation can be developed for each component in isolation, assured that their combination will in fact achieve the required system behaviors. For example, the system-wide constraint that passengers never

get farther from their destinations implicitly constrains both passengers and the elevator controller. In order to emerge with an implementation of the individual passengers and/or of the elevator controller, this constraint must be decomposed into constraints on the individual components. To reach one such decomposition, consider splitting the original constraint into the following two constraints:
C1: passengers whose destinations lie in different directions should not be in the same elevator, and
C2: an elevator with passengers inside should not move away from any of those passengers' destinations.
The combination of *C1* and *C2* satisfies the original constraint. Notice that *C2* alone would ensure the original constraint (since passengers would never be moved further from their destinations), however *C1* is also needed for progress, to ensure that an elevator won't get deadlocked with passengers needing transportation in different directions inside.

Having done this decomposition, *C1* can be assigned as the responsibility of the passengers, and *C2* as the responsibility of the elevator controller, bringing us a step closer towards an implementation. Of course, *C1* requires further decomposition in order to emerge with constraints on individual passengers rather than on passengers as a group. This snapshot of the development process raises several issues:

- What does it mean to say that a constraint is the responsibility of some *subset* of the system's components? We answer this fully in Sect. 5.1. Briefly, only the responsible components should need to limit their activities in order to meet the constraint. Referring back to the package router example, we might say that the constraint for correct routing of packages is the responsibility of the router, *not* the arriving packages: packages arriving at the source must be left with complete freedom to have any of the bins as their destinations — we would not be very happy with an implementation of package router that could route packages correctly provided that all the arriving packages always had the leftmost bin as their destination!
- Individual components may need access to information about other components in order to meet their assigned constraints. For example, in order to meet constraint *C2* the elevator controller must know the direction toward the destinations of the elevator's passengers. This need for information may induce the need for interfaces and associated protocols of use between the components. Continuing the example, the floor buttons inside an elevator are there to allow the passengers to indicate their destination floors to the elevator controller (from which the controller can deduce which way to move the elevator). This is examined further in Sect. 5.2.
- There need not be a unique way of decomposing system wide constraints. Typically there will be a choice of decompositions, each with its own information needs, and hence its own set of interfaces among components. Choice of the 'best' implementation will take into account properties of these interfaces (e.g., cost, reliability). Indeed, the interface needs may even motivate reconsideration of the system constraints. For example, if we are prepared to weaken the constraint that passengers never get farther from their destinations, we can derive an elevator system with a simpler interface, but less expedient transportation of passengers to their destinations — a particularly dumb elevator could repeatedly move from bottom floor to top floor, and back down

again, stopping at every intermediate floor; passengers could board the elevator and stay on board until it finally reached their destination floor. The only information passengers would need is to be able to recognize when they have arrived at their destination floor. Choice of decompositions (of constraints) is an instance of a more general question of design tradeoffs, and goes beyond the brief of this paper; the reader is referred to [17, 18] for further discussion of our explorations in this area.

5.1 Assigned Responsibility for Constraints

I first introduce the notion of 'component', and then address what it means for a constraint to be assigned as the responsibility of a component.

Components: The notion of 'component' is an addition to the Gist language features described so far. The components of a specification are some some subset of the specification's objects. By declaring a type to be a component, all instances of that type are declared as components (e.g., by declaring type passenger to be a component, every passenger object would be declared to be a component). New objects can be introduced to serve as components (e.g., the elevator controller would be introduced and declared as a component).

Broadly speaking, each of the activities of the specification (demons and top level statements) is associated with one component object. This has the effect of ascribing each primitive change done by an activity (when it contributes that primitive change to a transition from one state to the next) as having being done by the associated component. For example, if the elevator-activity demon is associated with the elevator-controller component, then every primitive transition done by that demon, namely movement of an elevator between floors, or opening or closing of an elevator's doors, is ascribed as having been done by the elevator-controller component. In the case of passengers, there is a single passenger-activity demon, but we need to ascribe its primitive transitions to the component that is the particular passenger entering/exiting. To do this we make each passenger object a separate component, and associate the passenger with the corresponding instantiation of the demon on that same passenger. The result of this is that a passenger's entry into, or exit from, an elevator is ascribed as being done by that passenger.

In the elevator example, it turns out there is the need to distinguish passenger appearance from passenger interactions with elevators — passenger appearance will be left unconstrained (akin to the package router wherein packages are allowed to arrive at any time with any bins as their destinations), while passenger interactions with elevators are further constrained. Hence I introduce another component object, `passenger-appearance`, associated with the to level loop statement that creates a passenger at some floor with some destination. Those primitive changes are ascribed as having been done by the passenger-appearance component.

A straightforward way of indicating components and their associated activities is shown for the elevator specification, next:

```
type floor = 1..topfloor;
type elevator;
```

```
component type passenger;
component passenger-appearance;
component elevator-controller;

...relation definitions as before

loop while true
   do choose  ...demon statement as before
   activity-of (any passenger-appearance);

demon passenger-activity(p:passenger)
   when true
   do  ...demon statement as before
   activity-of p;

demon elevator-activity(e:elevator)
   when true
   do  ...demon statement as before
   activity-of (any elevator-controller);

...remainder of definitions as before
```

In the above, `component` is used to declare instances of type `passenger` to be components, and two new objects, `passenger-appearance` and `elevator-controller` to be components. The top-level loop statement is declared to be an activity of the `passenger-appearance` component, `elevator-activity` demon is declared to be an activity of an `elevator-controller` component (of which there will be only the one), and invocation of the `passenger-activity` demon on passenger `p` is declared to be an activity of that passenger component `p`.

The net result of all this is that every primitive change in each transition of each behavior is ascribed as being done by one or more components (more than one arises if the same primitive change is simultaneously contributed to the same transition, and those changes are ascribed as being done by several different components — then, each of those components is said to have done that change; this complication does not arise in the examples considered in this paper).

Assigned Responsibility: Intuitively, when a constraint has been assigned as the responsibility of particular components, only those components are to limit their activities to prune out behaviors violating the constraint.

Recall that pruning discards behaviors from the set of possible behaviors to retain only those that satisfy the constraints. Consider pairs of behaviors, one from the set of behaviors retained by pruning, one from the set of behaviors discarded by pruning. They will diverge at some state, where up to that state they have performed exactly the same transitions, but their transitions emerging from that state differ (i.e., have a different set of primitive changes). We say that pruning is achievable by a set of components if for every such pair of behaviors (one retained, one discarded), the transitions at their point

Fig. 3. Assigned responsibility pruning

Suppose that there is a constraint that prohibited passengers whose destinations lie in different directions from being in the same elevator, and further suppose that passengers `p1` and `p2` have destinations in different directions. Then:

- Under the plain style of constraint pruning (without considering assigned responsibility), this would discard the behavior(s) including transition *t4*, because it leads to a state violating the constraint.
- If, however, the constraint were assigned as the sole responsibility of passenger `p1`, then discarding only the behavior(s) including the transition *t4* would rely upon component `p2`, which is *not* responsible, to make the only difference between the retained transition *t2* and the discarded transition *t4*. Instead, to meet the responsibility, pruning must discard not only transition *t4*'s behaviors, but also transition *t2*'s behaviors. This works because the responsible component, `p1`, does something different between every pair of discarded and retained behaviors, namely, enters, or does not enter, the elevator. Intuitively, `p1`, the sole responsible component, cannot risk entering the elevator because to do so relies upon `p2` to choose to not enter at the same time.

Note that in order to prune to meet assigned responsibility constraints, it may be necessary (as in the small example) to discard more than just the behaviors that violate the constraint.

Outline of Definition of Assigned Responsibility Pruning: Here follows a brief outline of the definition of pruning of behaviors when constraints have been assigned responsibility.

- *Assigned responsibility pruning:* Given a set of behaviors, a *pruning* is some subset of those behaviors. *Assigned responsibility pruning* of a set of behaviors is the largest *acceptable* pruning of that set.
- *Acceptable:* A pruning is *acceptable* if, at every state in the tree of pruned behaviors, pruning is *pointwise acceptable*.
- *Tree of behaviors:* A set of behaviors may be regarded as a *tree* of behaviors, by sharing common initial segments of those behaviors (recall that a behavior is a sequence of states separated by transitions).
- *Pointwise acceptable:* At a state in a tree of pruned behaviors, pruning is *pointwise acceptable* if every transition emerging from that state is *distinguishable* from every transition *omitted* at that state by the components *responsible* at that state.
- *Omitted:* A transition is *omitted* at a state within a tree of pruned behaviors if it is is not among the transitions emerging from that state, but is among the transitions emerging from the corresponding state in the tree of unpruned behaviors.
- *Distinguishable:* Two transitions are *distinguishable* by a set of components if their sets of primitive changes differ with respect to the changes ascribed to one or more of those components (recall that a transition comprises a set of primitive changes, and each primitive change is ascribed as having been done by some component).
- *Responsible at a state:* The set of components *responsible at a state* is the union of the sets of components *responsible* for behaviors *pruned at that state*.
- *Responsible:* The set of components *responsible* for a behavior is the union of the sets of components assigned responsibility for the constraints violated by that behavior.

– *Pruned at a state:* A behavior is *pruned at a state* if that behavior is included in the unpruned behaviors, but not included in the pruned behaviors, and the state is the lowest (latest) state in the behavior that is among the states within the tree of pruned behaviors.

For a more extended discussion of the definition and properties of such pruning, see [14].

5.2 Portion of Development of the Elevator System

I illustrate some of the intertwining between constraints and interfaces by showing a portion of the early stages of the development from the elevator specification towards an implementation.

My claim is that the interfaces present in typical elevator systems are there to provide information so as to permit the components to meet their responsibilities. For example, the presence of buttons on each floor to summon elevators is an interface to pass information from passengers to the elevator controller; the direction lights that indicate which way an elevator will move are present to pass information from the elevator controller to passengers. These interfaces arise as part of the development process from closed system specification toward specifications of the individual components, and ultimately their implementations.

To investigate this, I have rationalized the design of existing elevator systems by starting from the Gist specification of Sect. 4, incrementally decomposing the system-wide constraints into pieces which can be assigned as the responsibility of individual components, determining the information that each component needs to live up to its responsibilities, and deducing possible interfaces that provide such information. A portion of this development process is shown next; the complete development is to be found in [14]. As will be obvious, my decompositions are done in an ad-hoc manner; for a more organized approach to this (and related) activity, see [11].

1. The initial constraints defining suitably rapid transportation are:
   (a) `no-farther-from-destination` – a passenger must never move further from his/her destination floor.
   (b) `no-delay-to-riders` – passengers riding inside elevators must not be unnecessarily delayed. 'Unnecessary delay' can be defined on a history as a contiguous sequence of states during which a passenger was inside the elevator while the elevator remained inactive (didn't move, open or close its doors, or take on or let off passengers).

   These constraints are initially assigned as the joint responsibility of the controller and all passengers.
2. Decompose[1] `no-farther-from-destination` by:
   (a) Defining the (single-valued) Passenger Direction (`P-D`) of a passenger to be the direction (up or down) in which that passenger must go to reach his/her destination floor. (More precisely, the `P-D` of a passenger will have no value when

[1] To be valid, a decomposition of a constraint must result in a specification whose set of behaviors is a subset of the behaviors of the original specification

the passenger is at his/her destination floor, so it is either single-valued or has no value.) *This definitional step names a piece of information in preparation for future steps.*

(b) Choosing the implication of `no-farther-from-destination`, that all riders in a moving elevator have the same `P-D` values, to become the explicit constraint `riders-in-moving-elevator-compatible`. This is assigned as the responsibility of the controller and all passengers.

(c) Using the introduced constraint to simplify `no-farther-from-destination`; its simplified form is that a moving elevator with a rider must be moving in that rider's `P-D` direction.

(d) Assigning the simplified form of `no-farther-from-destination` as the responsibility of only the elevator controller. The constraint is renamed accordingly to `move-in-rider's-P-D`.

The above steps show how one of the initial system-wide constraints, `no-farther-from-destination`, is decomposed, and the resulting pieces assigned as the responsibility of individual components within the system. Continuing this process eventually leads to the behavior and interface typical of many elevator systems.

5.3 Summary of initial stages of implementation development

We have seen how the initial stages in the development from a system-wide Gist specification address the decomposition of the 'closed system' specification into specifications of the individual components. System-wide constraints are decomposed in order that their pieces can then be assigned as the responsibility of the individual components. This results in specifications of the individual components from which implementations can thereafter be developed independently, assured that their combination will achieve the behaviors desired of the system. In the course of this development, the interfaces between components and associated protocols of use of those interfaces emerge as those required to provide components with the information they will need to meet their individual specifications.

The need for this decomposition arises from Gist's encouragement of the closed system style of specification, together with its generate and test way of expressing behaviors. For systems that interact in an ongoing and non-trivial manner with their environments (of which the package router and the elevator controller are simple examples), it is often clearest to specify them as closed systems, and proceed with development from that point.

6 Related Work

6.1 Gist-specific Related Work

Gist was developed primarily by Bob Balzer, Neil Goldman and David Wile at ISI, based on the principles that they first established for such a language [3] and their belief in the development of software by transformation from specifications [4]. A considerable amount of research related to specification issues has been done by these people, and the other

past and present members of Bob Balzer's Software Sciences Division at ISI: Dennis Allard, Bob Balzer Kevin Benner, Don Cohen, Wellington Chiu, Lee Erman, Michael Fehling, Steven Fickas, Neil Goldman, Lewis Johnson, Yingsha Liao, Philip London, Matthew Morgenstern, Jay Myers, K Narayanaswamy, Bill Swartout, David Wile and Kaizhi Yue.

Our group's experiences with Gist, and companion efforts work on the software development process, are summarized in [2]. Briefly, we have found the following issues to hold for Gist specifications of complex systems (and, we believe, hold for all formal specifications, regardless of the language):

Hard to read — whatever the formalism, newcomers unfamiliar with it have a hard time understanding what it means. We have experimented with tools that automatically paraphrase Gist specifications in English [32, 33]. See also the **hard to write** item for further methods to present complex specifications.

Hard to analyze — discovering and/or proving properties of Gist specifications is quite difficult. We have built a symbolic evaluator to explore the dynamic implications of Gist specifications, and hooked this to the paraphraser (mentioned above) to present these to the reader [9]. Scaling up symbolic evaluation to large specifications is problematic, however. Recently we have been exploring simulation together with abstraction — essentially, given a question we would like to ask of the behaviors denoted by a specification, we abstract from the fully detailed specification to get a smaller version with only those aspects relevant to the question, and then employ simulation of that smaller version to find the answer [6].

Hard to write — large, complex systems have large and complex specifications, in spite of the advantages provided by specification languages. Incremental development of specifications is one way to mitigate the difficulty of construction, explanation and modification of such specifications [20]. 'Evolution transformations' are our transformational technology to support this process - they are transformations designed to deliberately change the meaning of the specification to which they are applied [15, 24].

Hard to implement — we are not able to transform an arbitrary Gist specification into an implementation. Gist's wide range of powerful constructs is quite hard to implement in general, although it is clear what the major subtasks of such a process must achieve [27]. We have explored the transformational implementation of some subsets of these properties, for example, temporal reference to past states ("historical" reference) is amenable to transformational implementation [16]. Another approach that we have taken is to extract some of Gist's features, notably the relational database together with demons and a variation on the notion of constraints, and layer these on top of a conventional programming language, Common Lisp (we are also working on doing the same on top of Ada) [10]. These ideas are incorporated into a sizable programming environment that our group uses on a day-to-day basis [19].

We have built an experimental environment to support requirements acquisition and specification construction; within this, Gist forms the core of the common knowledge representation language for expressing the requirements and the emerging specifications [25, 26].

6.2 Other Related Work

Other work on 'behavior oriented specification' includes Milner's CCS calculus [28, 29], essentially a calculus for reasoning about trees of behaviors; Dijkstra's guarded commands provide a well-grounded basis for describing and reasoning about distributed systems [12]; based on this is the 'action systems' work [1], somewhat similar in style and purpose to our Gist efforts. The language ERAE [21] is also similar to Gist in terms of constructs and approach to specification.

Individual features of Gist derive from previous work:

- The relational data base model - espoused by [31].
- Temporal logic, at least in its use to talk and reason about the past. Gist's use of historical reference is very close to the approach of Sernadas in his temporal process specification language, DMTLT ([30]).
- Automatic demon invocation - seen in the AI languages PLANNER and Qlisp ([8]).
- Non-determinism in conjunction with constraints - closest to non-deterministic automata theory, [23].
- Operational semantics and closed system assumptions - as seen in simulation languages, [7, 35]), and overviewed in [36].

Since the development of Gist, there have been some continuations of some closely related ideas:

- Formal semantics for assigning constraints as the responsibility of particular components are given in terms of a deontic logic [13].
- The role of 'responsibility' during system design has been studied further: [17, 18].
- A model to support the acquisition of requirements, leading through a Gist-like specification towards an implementation, has been proposed and studied [11].

7 Conclusions and Acknowledgements

These notes have attempted to give a feel for Gist, a specification language designed to facilitate the expression of systems exhibiting complex, ongoing behaviors. There is a growing consensus that in order to achieve major improvement in software production and maintenance, the entire programming process must be formalized and given machine support (see, for example, the joint report [5]). The keystone of such an approach is the formal specification of the requirements of the task to be programmed, for which purpose a specification language tailored for ease of expression of such requirements is essential.

Gist has been developed by members of Bob Balzer's Software Sciences Division at ISI, supported by the Defense Advanced Research Projects Agency, Rome Air Development Center, and the National Science Foundation. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, RADC, NSF, the U.S. Government, or any other person or agency connected with them. Thanks are due to Prof. Dr. Bernhard Möller, for his careful scrutiny of earlier drafts of this document.

References

1. R.J.R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM TOPLAS*, 10(4):513–554, 1988.
2. R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1267, November 1985.
3. R. Balzer and N. Goldman. Principles of good software specification and their implications for specification languages. In *Specification of Reliable Software*, pages 58–67. IEEE Computer Society, 1979.
4. R. Balzer, N. Goldman, and D.S. Wile. On the transformational implementation approach to programming. In *Proceedings, 2nd International Conference on Software Engineering, San Francisco, California*, pages 337–344, October 1976.
5. R. Balzer, T.E. Cheatham Jr., and C. Green. Software technology in the 1990's: Using a new paradigm. *Computer*, pages 39–45, November 1983.
6. K. Benner. ARIES Simulation Component (ASC) demonstration. In *Proceedings of the 7th KBSE Conference*, page 257, McLean, VA, September 1992. IEEE Computer Society Press.
7. G.M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Auerbach, 1973.
8. D. Bobrow and B.Raphael. New programming languages for artificial intelligence. *ACM Computing Surveys*, 6(3):153–174, September 1974.
9. D. Cohen. Symbolic execution of the Gist specification language. In *Proceedings, 8th International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany*, pages 17–20, August 1983.
10. D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 225–234. ACM Press, 1989. SIGMOD RECORD Volume 18, Number 2, June 1989.
11. A. Dardenne, S. Fickas, and A. van Lamsweerde. Goal-directed concept acquisition in requirements elicitation. In *Proceedings, 6th International Workshop on Software Specification and Design, Como, Italy*, pages 14–21. IEEE Computer Society Press, 1991. A substantially expanded version is to appear in Science of Computer Programming.
12. E.W. Dijkstra. *A discipline of programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
13. E. Dubois. A logic of action for supporting goal-oriented elaborations of requirements. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, USA*, pages 160–168. Computer Society Press of the IEEE, 1989.
14. M.S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2):198–234, April 1987.
15. M.S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989.
16. M.S. Feather. Transformational implementation of historical reference. In B. Möller, editor, *Constructing Programs from Specifications*, pages 225–242. North-Holland, 1991. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May 1991.
17. M.S. Feather, S. Fickas, and B.R. Helm. Composite system design: the good news and the bad news. In *Proceedings of the 6th Annual RADC Knowledge-Based Software Engineering (KBSE) Conference, Syracuse, NY, September 1991*, pages 16–25. IEEE Computer Society Press, 1991.
18. S. Fickas and B.R. Helm. Knowledge representation and reasoning in the design of composite systems. *IEEE Transactions on Software Engineering*, 18(6):470–482, June 1992.

19. N. Goldman and K. Narayanaswamy. Software evolution through iterative prototyping. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia*, 1992.
20. N. M. Goldman. Three dimensions of design development. In *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pages 130–133, August 1983.
21. J. Hagelstein. Declarative approach to information system requirements. *Journal of Knowledge-Based Systems*, 1(4):211–220, September 1988.
22. G. Hommel. Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage. Technical Report PDV-Report, KfK-PDV 186, Part I, Kernforschungszentrum Karlsruhe GmbH, August 1980.
23. J.E. Hopcroft and J.D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley, 1969.
24. W.L. Johnson and M.S. Feather. Using evolution transformations to construct specifications. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, 1991.
25. W.L. Johnson, M.S. Feather, and D.R. Harris. Integrating domain knowledge, requirements and specifications. *Journal of Systems Integration*, 1:283–320, 1991.
26. W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, October 1992.
27. P.E. London and M.S. Feather. Implementing specification freedoms. In C. Rich and R. Waters., editors, *Readings in Artificial Intelligence and Software Engineering*, pages 285–305. Morgan Kaufmann, 1986. Originally published in Science of Computer Programming, 1982 No. 2, pp 91-131.
28. R. Milner. *A calculus of communicating systems*, volume 92 of *Lecture notes in computer science*. Springer-Verlag, 1980.
29. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
30. A. Sernadas. Temporal aspects of logical procedure definition. *Information Systems*, 5(3):167–187, 1980.
31. J. Smith and D. Smith. Database abstractions: aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.
32. W. Swartout. Gist english generator. In *Proceedings, AAAI-82*, pages 404–409, August 1982.
33. W. Swartout. The Gist behavior explainer. In *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pages 402–407, August 1983.
34. W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, 1982.
35. P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, May 1982.
36. P. Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, 1984.

This article was processed using the LaTeX macro package with LMAMULT style